Functions – Return Values

- Returns None
  - Output not reusable

```
1 def fxn_avg (a, b):
2    sum = a + b
3    avg = sum/2
4    print (avg)
5
6 y = fxn_add (10, 10)
7 print (y) # None
8 type(y)
9 # <class 'NoneType'>
```

- Returns a value
  - Output reusable

```
1 def fxn_avg (a, b):
2    sum = a + b
3    avg = sum/2
4    return avg
5
6 y = fxn_add (10, 10)
7 print (y) # 10
8 type(y)
9 # <class 'float'>
```

Functions – Return Values

- Only ONE value returned
  - Comma seperated values implicitly converted into tuple

```
1 def fxn_avg (a, b):
2    sum = a + b
3    avg = sum/2
4    return sum, avg
5
6 y = fxn_add (10, 10)
7 print (y) # (20, 10.0)
8 type(y) # <class 'tuple'>
```


Functions – Duplication

- Adding arbitrary numbers

```
1 def fxn_add (*args):
2    sum = 0
3    for i in args:
4        sum += i
5    return sum
6
7 fxn_add (1,2,3)
```

- Average of arbitrary numbers

```
1 def fxn_avg (*args):
2    sum = 0
3    count = len(args)
4    for i in args:
5        sum += i
6    avg = sum/count
7    return avg
8
7 fxn_avg (1, 2, 3)
```

Functions – Reuse& Deduplication

```
1 def fxn_add (*args):
2    sum = 0
3    for i in args:
4        sum += i
5    return sum
6
7 fxn_add (...)
8
```

```
1 def fxn_avg (*args):
2    sum = fxn_add(*args)
3    count = len(args)
4    avg = sum/count
5    return avg
6
7 fxn_add ()
8
```


Functions – Reuse& Deduplication

- Without reuse

```
1 def fxn_avg (*args):
2    sum = 0
3    count = len(args)
4    for i in args:
5        sum += i
6    avg = sum/count
7    return sum
8
9 fxn_avg (...)
```

- With reuse

```
1 def fxn_avg (*args):
2    sum = fxn_add(*args)
3    count = len(args)
4    avg = sum/count
5    return avg
6
7 fxn_add ()
```


Functions – More on Reuse

- Module files

```
# csc1017f.py

1 def fxn_1 ():
3    return "fxn_1"
4
5 def fxn_2 ():
6    return "fxn_2"
7
8 def fxn_3 ():
9    return "fxn_3"
```

- Module reuse

```
1 import csc1017f
2
3 csc1017f.fxn_1()
4 csc1017f.fxn_2()
5
5 from csc1017f import fxn_3
6
7 fxn_2()
6
7 fxn_add ()
```


STOP N – Examples with Builtins

- Would this work? What is the result? What is the return type?
  - `y = print("hello")`
  - `y = len("four")`
  - `y = input("Enter value:")`

UCT Department of Computer Science
Computer Science 1017F

Testing

Lighton Phiri <lphiri@cs.uct.ac.za>
April 2015

Introduction

- What is an error?
  - When your program does not behave as intended or expected.
- What is a bug?
  - "...a bug crept into my program ..."
- Debugging
  - the art of removing bugs

Types of Errors – Syntax Errors

- Syntax Errors
  - Failure to conform to Pythonic syntactic rules
  - Improper use of Python language – usually Syntax Errors.
    - 12var = 2

```
File "<stdin>", line 1
12var = 2
    ^
SyntaxError: invalid syntax
```

- Fixing Syntax Errors
  - Easiest type of error to fix
    - Conform! Follow the rules!

Types of Errors – Runtime Errors

- Runtime Errors
  - Program is syntactically correct but error detected after execution.
    - `a = 1/0 (ZeroDivisionError: division by zero)`
    - `a = b + 10`
    - `3 “four”[100] (IndexError: string index out of range)`
    - `I/O operations`

- Fixing Runtime Errors
  - Exception handling.
  - Normally able to anticipate potential program flaws
    - Examples above illustrate this

Types of Errors – Runtime Errors

- Exception handling involved

```
1 def fxn_divide (a, b):
2    result = 0
3    try:
4       result = a / b
5    except ZeroDivisionError:
6       print ("Division by ZERO error!")
7    return result
8
9 fxn_divide(1, 0)
```


Types of Errors – Logical Errors

- Logical Errors
  - Program parsed by by interpreter and runs successfully, but produces undesirable results
    - `sum = 1 - 1`
    - `Conditionals—logical operators—and, or, not`

- Fixing Logical Errors
  - Programmer responsibility to ensure code functions as required before releasing it
    - Debugging
    - Testing

STOP 1: Identifying Errors

```
1 def add_x (a, b):
2    return a + b
3    print (“Output”) # Logical Error
4
5 add_x(“1”, “2”) # Logical Error
6 add_x(+) # Syntax Error
7 add_x() # Runtime Error
```

- Identify errors in code snippet—what types are they?
- An individual recently won a $5k Google bounty for figuring out that YouTube videos could easily be deleted. What error was identified?

Debugging

- **Debugging** is the process of finding **errors** or **bugs** in the code.
- Debugging techniques
  - Application stacktraces—Android app, Web browsers, OSes
- A **debugger** is a tool for executing an application where the programmer can carefully control execution and inspect data.
- Features include:
  - step through code one instruction at a time
  - viewing variables
  - insert and remove breakpoints to pause execution

Basic Debugging Techniques

- Commenting out code thought to be root cause of problem

```
1 # result = a/b # possible division by 0
2 if b > 0
3    result = a/b
4
```

- Builtin output function
  - print (...)

Debugging with Wing

- Hands on Laboratory Exercise
  - Next week
- Set breakpoints in the code to halt execution path
- Initiate execution in debug mode, start debugging
- Facilitate user input/output in I/O Debug window
- Step through code and view—in real-time—variable change in stack data window
- Stop debugging when error is located